A Compressed-Gap Data-Aware Measure

Nicola Prezza[1]

Department of Mathematics and Computer Science, University of Udine, Italy
prezza.nicola@spes.uniud.it

Abstract. In this paper, we consider the problem of efficiently representing a set S of n items out of a universe $U = \{0, ..., u-1\}$ while supporting a number of operations on it. Let $G = g_1 ... g_n$ be the gap stream associated with S, *gap* its bit-size when encoded with *gap-encoding*, and $H_0(G)$ its empirical zero-order entropy. We prove that (1) $nH_0(G) \in o(gap)$ if G is highly compressible, and (2) $nH_0(G) \leq n\log(u/n) + n \leq uH_0(S)$. Let d be the number of *distinct* gap lengths between elements in S. We firstly propose a new space-efficient zero-order compressed representation of S taking $n(H_0(G)+1) + \mathcal{O}(d\log u)$ bits of space. Then, we describe a fully-indexable dictionary that supports *rank* and *select* queries in $\mathcal{O}(\log(u/n) + \log\log u)$ time while requiring asymptotically the same space as the proposed compressed representation of S.

Keywords: dictionary problem, gap encoding, entropy, compression, rank, select

1 Introduction and Related Work

The *dictionary problem* on *set data* asks to maintain a (space-efficient) data structure called *indexable dictionary* over a set $S = \{s_1, ..., s_n\} \subseteq \{0, ..., u-1\} = U$, $s_1 < s_2 < ... < s_n$, supporting efficiently a range of queries on S. In this problem, U is an ordered set and is called *universe*. As showed by Jacobson in his doctoral thesis [10], a set of just two operations, *rank* and *select*, is sufficient and powerful enough in order to derive other fundamental functionalities desired from such a structure: *member*, *successor*, and *predecessor*. $rank(S, x)$, with $x \in U$, is the number of elements in S that are smaller than or equal to x. $select(S, i)$, where $0 \leq i < n$, is the i-th smallest element in S. In this paper, we focus on *fully-indexable dictionaries (FIDs)*, i.e. data structures supporting both rank and select operations efficiently.

Jacobson in [10] proposed a solution for this problem taking $u + o(u)$ bits of space and supporting constant-time rank. Constant-time select within $o(u)$ bits of additional space was added by Munro [13] and Clark [5]. These results were further improved firstly by Pagh [14] (who considered *rank*) and then by Raman et al. [16] (*rank* and *select*) with structures having the same time complexities and requiring only $\mathcal{B}(n, u) + \mathcal{O}(u\log\log u/\log u)$ bits of space, where $\mathcal{B}(n, u) = \lceil\log\binom{u}{n}\rceil$ is the minimum number of bits required in order to distinguish *any* two size-n subsets of U. Finally, Pătraşcu [15] reduced the sublinear term to $\mathcal{O}(u/polylog(u))$ while retaining constant query times. Despite these last results being optimal for big values of n, the $o(u)$

term can however be much bigger than $\mathcal{B}(n,u)$ (even exponentially) if n is very small. Moreover, even the $\mathcal{B}(n,u)$ term is not optimal for *all* instances, and can be improved in many cases of practical interest. To see why this fact holds true, it is sufficient to notice that zero-order entropy compressors encode to the same bit-size *all* size-n subsets S of U, without taking advantage of the structure of S (for example, long or regular distances between its elements). This problem motivates the search for more *data-aware* measures able to break the $\mathcal{B}(n,u)$ limit in some cases. One of the most widely known such data-aware measures is *gap* [3], which is defined to be the sum of all bit-lengths of the distances between consecutive elements in S. If these distances are not evenly distributed, *gap* can be much smaller than $\mathcal{B}(n,u)$, reaching 10%-40% of $\mathcal{B}(n,u)$ in some instances of practical interest [9]. By using *logarithmic codes* such as Elias δ-encoding [6], the stream of gaps can be compressed to $gap + o(gap)$ bits, where the $o(gap)$ overhead comes from the prefix property of such codes, needed to unambiguously reconstruct codeword boundaries. In [9], Gupta et al. show how to build a FID based on δ-encoding requiring only $gap + \mathcal{O}(n \log(u/n)/\log n) + \mathcal{O}(n \log\log(u/n))$ bits of space and supporting rank and select in $AT(u,n) \in o((\log\log u)^2)$—this is nearly optimal within that space, see [1,2]—and $\mathcal{O}(\log\log n)$ time, respectively. Other recent works [11,17] showed that constant-time queries can be supported using $gap + \mathcal{O}(n \log\log(u/n)) + o(u)$ bits of space, where the $o(u)$ term is $\mathcal{O}(u \log\log u/\sqrt{\log u})$ in [11] and $\mathcal{O}(u \log\log u/\log u)$ in [17].

gap reaches its maximum when all gap lengths are equal. However, it is clear that in this scenario other techniques (e.g. zero-order entropy compression) could be flanked to gap encoding in order to turn this worst-case into a $\mathcal{O}(n)$-bits best-case. In this paper we explore the possibility of compressing the stream of gaps G to its zero-order empirical entropy $H_0(G)$, aiming at obtaining $nH_0(G)$ as leading term in the space complexity of our structures. Similar techniques are already employed in BWT-based text compression algorithms [4], where runs of zeros in the move-to-front encoding of the BWT are compressed using run-length-encoding followed either by zero-order entropy compression or by logarithmic encoding [6] (runs being mostly dominated by small numbers). We firstly observe that $nH_0(G) \in o(gap)$ if gaps are highly compressible, and prove that $nH_0(G)$ does not exceed $n \log(u/n) + n$ bits. This bound is provably smaller than the zero-order empirical entropy of the set S and of any of its decodable gap-encoded representations.

These considerations suggest that the data-aware measure $nH_0(G)$ should be preferred to *gap* in cases where the overhead introduced by the zero-order compressor (e.g. a codebook) is negligible. Our work goes in this direction. First of all, we show a new zero-order compressed representation of bitvectors taking $nH_0(G) + n + \mathcal{O}(d \log u) \leq uH_0 + n + \mathcal{O}(\sqrt{u} \log u)$ bis of space, where u is the length of the bitvector, H_0 its zero-order empirical entropy, n the number of bits set, and d the number of *distinct* distances between bits set. d is trivially upper-bounded by n and $\mathcal{O}(\sqrt{u})$, and is negligible in many practical cases (e.g. when S is dense or the gaps are evenly distributed).

We finally propose a fully-indexable dictionary that answers *rank* and *select* queries in $\mathcal{O}(\log(u/n) + \log\log u)$ time and whose space occupancy is of $(1 + o(1))nH_0(G) + (3 + o(1))n + \mathcal{O}((d + \log\log u)\log u)$ bits. In all cases where $H_0(G) \in \omega(1)$ and $d \geq \log\log u$, this is asymptotically the same space as our new bitvector representation. Moreover, if S is dense enough—$n \in \Omega(u/polylog(u))$—all queries are supported in $\mathcal{O}(\log\log n)$ time, which is optimal within this space.

2 Gap-Encoded Dictionaries

In this section we will assume that $u - 1 \in S$, so that each gap corresponds to an element in S (i.e. the element following the gap). If $u - 1 \notin S$, then we can simply use an extra bit to denote this case and encode the final gap length separately. We will moreover assume that $n \leq u/2$. Logarithms are taken in base 2, unless differently specified. In *gap encoding*, we represent the set $S = \{s_1, ..., s_n\} \subseteq \{0, ..., u - 1\} = U$, $s_1 < s_2 < ... < s_n$ as the stream of gaps $g_1, ..., g_n$, where $g_1 = s_1 + 1$ and $g_i = s_i - s_{i-1}$ for $i > 1$. In order to reduce space occupancy of the stream, variable-length encoding can be used to encode each of the g_i. The data-aware measure $gap(S)$ is defined as $gap(S) = \sum_{i=1}^{n} \big(\lfloor \log g_i \rfloor + 1\big)$, that is, the total number of bits required in order to store all g_i's using the *minimum* number of bits to represent each gap. When clear from the context, we will simply write *gap* instead of *gap*(S). Clearly, S cannot be represented using only *gap* bits since we need additional information in order to make the stream uniquely decodable. We adopt a notation similar to [9] and indicate with $Z_{\mathcal{C}}(S)$—or simply $Z_{\mathcal{C}}$ when clear from the context—the decoding overhead (in bits) introduced by the coding scheme $\mathcal{C}$. If we use a separate bitvector B marking with a 1 the beginning of each code, then we obtain $Z_B = gap$. Another solution is to use *logarithmic codes* such as Elias γ or δ-encoding [6]. In γ-encoding, we encode $\lfloor \log g_i \rfloor + 1$ in unary, followed by the $\lfloor \log g_i \rfloor$-bits binary representation of g_i without the most significant 1. Then, $Z_\gamma = gap - n$. A better solution is δ-encoding, where we encode with γ the number $\lfloor \log g_i \rfloor + 1$, followed by the $\lfloor \log g_i \rfloor$-bits binary representation of g_i without the most significant 1. Then, $Z_\delta = 2\sum_{i=1}^{n} \lfloor \log(\lfloor \log g_i \rfloor + 1) \rfloor$ bits. log being a concave function, the worst-case of *gap* occurs when $g_1 = g_2 = ... = g_n = u/n$ (by Jensen's inequality), yielding the upper bounds $gap \leq n\log(u/n) + n$ and $Z_\delta \leq 2n\log(\log(u/n) + 1)$. Then, one can prove the following (for the original proof, see [8]):

Lemma 1. *$gap \leq \mathcal{B}(n, u)$ if $n \leq u/2$.*

Proof. The claim follows directly from $gap \leq n\log(u/n) + n$ and from the fact that $\mathcal{B}(n, u) = n\log(u/n) + n\log e - \Theta(n/u) + \mathcal{O}(\log n)$ if $n \leq u/2$ □

Moreover, let $H_0(S) = \frac{n}{u}\log(\frac{u}{n}) + \frac{u-n}{u}\log\frac{u}{u-n}$ be the zero-order empirical entropy of the set S. Since $\mathcal{B}(n, u) \leq uH_0(S)$, we have that:

Corollary 1. *$gap \leq uH_0(S)$ if $n \leq u/2$.*

The above inequalities are important as they show that gap encoding can never perform worse than zero-order entropy compression. On the other hand, experiments show [9] that *gap* can be significantly smaller than $\mathcal{B}(n, u)$ for many cases of interest, thus motivating its use in practical applications. In the following section we take one step forward, exploring what happens when we treat S as a sequence on the alphabet $\{g_1, ..., g_n\}$ and then apply zero-order entropy compression to it.

2.1 A Compressed-Gap Data-Aware Measure

gap reaches its worst-case of $n \log(u/n) + n$ bits when all gaps have the same length. However, it is clear that entropy compression should turn this worst-case scenario into a best-case, since the zero-order empirical entropy of such a configuration is equal to 0. More formally, let's consider the following representation G of S. We define G to be the sequence $g_1 g_2 ... g_n \in \Sigma_{gap}^n$, where $\Sigma_{gap} = \{g_1, g_2, ..., g_n\}$. Let moreover $d = |\Sigma_{gap}|$ be the alphabet size and $f(s) = occ(s)/n$, $s \in \Sigma_{gap}$, be the empirical relative frequency of s in G, where $occ(s)$ is the number of occurrences of s in G. We define the *zero-order empirical entropy of the gaps* $H_0(G)$ to be

Definition 1. $H_0(G) = -\sum_{s\in\Sigma_{gap}} f(s) \log (f(s))$

$nH_0(G)$ is the minimum number of bits output by any compressor that encodes G assigning a unique code to each symbol in Σ_{gap}. First of all, we observe that $nH_0(G)$ can be significantly smaller than *gap*: if $g_1 = g_2 = ... = g_n = u/n$, then $n \log(u/n) \leq gap \leq n \log(u/n)+n$ and $nH_0(G) = 0$. Moreover, $nH_0(G)$ is never worse than the length of any decodable gap-compressed sequence:

Lemma 2. *$nH_0(G) \leq gap + Z_{\mathcal{C}}$, where $\mathcal{C}$ is any prefix coding scheme.*

Proof. Follows directly from the fact that no prefix code can compress G in less than $nH_0(G)$ bits. □

Using Lemma 2 and the bounds for *gap* and Z_δ derived in the previous section, one can obtain $H_0(G) \leq \log(u/n) + 2\log(\log(u/n) + 1) + 1$. With the following theorem we show a much stronger upper bound:

Theorem 1. $H_0(G) \leq \log(u/n) + 1$

Proof. We want to compute

$$\max_{\Sigma_{gap}\subseteq\mathbb{N}_{>0}} \max_{f:\Sigma_{gap}\rightarrow\mathbb{R}^+} H_0(G)$$

where the alphabet Σ_{gap} and the empirical frequency function f must satisfy:

$$n \sum_{s\in\Sigma_{gap}} f(s) \cdot s = u \tag{1}$$

Let $d = |\Sigma_{gap}|$. From Definition 1 and from the concavity of log, we have that $H_0(G)$ reaches its maximum $H_0(G) = \log d$ when all frequencies are equal, i.e. $f(s) = d^{-1}$ for all $s \in \Sigma_{gap}$. We thus have

$$\max_{\Sigma_{gap} \subseteq \mathbb{N}_{>0}} \max_{f:\Sigma_{gap} \to \mathbb{R}^+} H_0(G) = \max_{\Sigma_{gap} \subseteq \mathbb{N}_{>0}, f(s)=d^{-1},\ s \in \Sigma_{gap}} \log d$$

In order to maximize $\log d$, we now have to find Σ_{gap} of maximum cardinality that satisfies condition (1). It is easy to see that $\Sigma_{gap} = \{1, ..., d\}$ minimizes $\sum_{s \in \Sigma_{gap}} s = \sum_{i=1}^{d} i = d(d+1)/2$. Since, moreover, $f(s) = d^{-1}$ for all $s \in \Sigma_{gap}$, we can rewrite (1) as $nd^{-1}\big(d(d+1)/2 + k\big) = u$, where $k \geq 0$. Solving in d, we obtain the set of solutions

$$\mathcal{Z} = \left\{ \big(b \pm \sqrt{b^2 - 8kn^2}\big)/(2n) \mid b = 2u - n \ \wedge \ k \geq 0 \right\}$$

for which we have $\max \mathcal{Z} = (2u - n)/n$ when $k = 0$. This implies that $\Sigma_{gap} = \{1, ..., (2u - n)/n\}$ and $f(s) = n/(2u - n)$ for all $s \in \Sigma_{gap}$ maximize $H_0(G)$. Our claim follows:

$$H_0(G) \leq \log d \leq \log(2u/n) = \log(u/n) + 1$$

□

Interestingly, the two measures *gap* and $nH_0(G)$ are upper-bounded by the same quantity $n \log(u/n) + n$. This is not a trivial result since, differently from $nH_0(G)$, *gap* does not include information needed to reconstruct unambiguously codeword boundaries (even though $nH_0(G)$, in turn, does not include information—e.g. a codebook—needed to decode codewords). Using the same arguments of Lemma 1 and Corollary 1, we can moreover derive the bounds:

Corollary 2. $nH_0(G) \leq \mathcal{B}(n, u) \leq uH_0(S)$ *if* $n \leq u/2$

The pair $\langle U, S \rangle$ can be represented as a length-u bitvector B with n bits set. Let $H_0 = H_0(S)$ be the zero-order entropy of B and d be the number of *distinct* distances between bits set in B. Then:

Corollary 3. *There exists a zero-order compressed representation of B taking $n(H_0(G) + 1) + \mathcal{O}(d \log u) \leq uH_0 + n + \mathcal{O}(d \log u)$ bits of space.*

Proof. Can be easily obtained by compressing the gap sequence with Huffman-encoding and by applying Corollary 2.

Note that the number d of *distinct* distances between bits set of B is trivially upper-bounded by n and $\mathcal{O}(\sqrt{u})$ [1].

[1] Assume, by contradiction, that $d \in \omega(\sqrt{u})$. Then, the set Σ_{gap} of gaps that minimizes $\sum_{s \in \Sigma_{gap}} s$ is $\Sigma_{gap} = \{1, ..., d\}$, for which we obtain $\sum_{s \in \Sigma_{gap}} s = \Theta(d^2) = \omega(u)$. This is an absurd since the sum of all gaps cannot exceed u.

3 A Compressed-Gap FID

Let us now turn our attention to fully-indexable dictionary data structures. Our aim is to obtain a structure that takes asymptotically the same space as the representation described in Corollary 3.

Our strategy is the following: we use Elias δ-encoding and exploit its property of being an *asymptotically optimal universal code* [6] to encode the gap stream in $(1+o(1))nH_0(G)+n$ bits. We then build a two-levels structure atop of this representation to support rank and select queries. We adopt an approach similar to [9] and firstly describe a binary-searchable dictionary (BSD) that supports all queries in $\mathcal{O}(\log u)$ time. The BSD is finally used as building block for our final structure, which improves all query times to $\mathcal{O}(\log(u/n) + \log\log u)$ within the same space.

Let Σ_{gap} and $f : \Sigma_{gap} \rightarrow \mathbb{R}^+$ be the set of all gap lengths and the empirical frequencies associated with the gap stream, respectively, and consider an (arbitrary) ordering of the symbols $ord : \Sigma_{gap} \rightarrow \{1,...,d\}$, $d = |\Sigma_{gap}|$ (i.e. a bijection) such that if $ord(g_i) < ord(g_j)$ then $f(g_i) \leq f(g_j)$ for all $g_i, g_j \in \Sigma_{gap}$. Let $\delta(x),\ x > 0$ be the Elias δ code of the integer x. Then, we associate the code $code(g_i) = \delta(ord(g_i))$ to each gap length $g_i \in \Sigma_{gap}$. Being δ an asymptotically optimal universal code [6], the bit length l of the compressed stream $code(g_1)...code(g_n)$ is at most $(1+o(1))nH_0(G)+n$ bits[2]. In the following we assume to work under the word RAM model with word size $\Theta(\log u)$ bits, so that we can extract any $\mathcal{O}(\log u)$-bits block from a plain bitvector in constant time. We store the bit representations of the compressed gaps sequentially in a bitvector $C[0,...,l-1] = code(g_1)...code(g_n)$. An additional array $D[1,...,d]$ defined as $D[i] = ord^{-1}(i)$ (the codebook) is moreover built to permit the decoding of codewords. Note that, given the starting position of $code(g_i)$, $0 \leq i < n$, in the bitvector C, we can extract and decode $code(g_i) = \delta(ord(g_i))$ in $\mathcal{O}(1)$ time: firstly, we need to decode the γ-prefix of $\delta(ord(g_i))$. This can be done in $\mathcal{O}(1)$ time using two universal tables of $\mathcal{O}(2^{\log\log u}\log\log u) = \mathcal{O}(\log u \log\log u)$ bits each (one for the unary prefix and the other for the rest of the γ-prefix of the code). This gives us (i) the bit-length of the γ-prefix of $\delta(ord(g_i))$, and (ii) the bit-length of $ord(g_i)$ (without the most significant bit). We can then extract the bits of $ord(g_i)$ and access $D[ord(g_i)] = g_i$ in constant time. To improve readability, in the next sections we will implicitly make use of this strategy and—provided that we know the starting position of $code(g_j)$ in C—say *read gap* g_j instead of *extract and decode* $code(g_j)$.

3.1 A Binary-Searchable Dictionary

We divide the elements of $S = \{s_1,...,s_n\}$ into blocks of size $t = \lceil \log u \rceil$ (we assume for clarity of exposition that t divides n; the following arguments

[2] Even when $H_0(G) = 0$, with δ-encoding we spend *at least* 1 bit per symbol, thus the additional n term. The $o(nH_0(G))$ term comes from overhead introduced by δ-encoding, and in the *worst* case (n distinct gaps, $H_0(G) \in \Theta(n \log n)$) equals $\Theta(n \log\log n)$ bits.

can be easily adapted to the general case). For each block $\{s_{it+1}, ..., s_{(i+1)t}\}$, $i = 0, ..., n/t - 1$, we store explicitly the smallest element s_{it+1} and a pointer to the beginning of $code(g_{it+2})$ in the bitvector C [3]. These structures are sufficient to obtain our BSD. $select(S, i)$, $0 \leq i < n$, is implemented by accessing the $\lfloor i/t \rfloor$-th block and reading $i \mod t < t$ gaps in C starting from $g_{\lfloor i/t \rfloor t+2}$. Then,

$$select(S, i) = s_{\lfloor i/t \rfloor t+1} + \sum_{j=\lfloor i/t \rfloor t+2}^{i+1} g_j$$

$rank(S, x)$, $x \in U = \{0, ..., u - 1\}$, is implemented by binary-searching the blocks according to explicitly stored elements s_{it+1}, $i = 0, ..., n/t - 1$, and then by extracting gaps in the block of interest until we reach element x. More formally, let $0 \leq i \leq n/t - 1$ be the biggest integer (if any) such that $s_{it+1} \leq x$. i can be found by binary search in $\mathcal{O}(\log u)$ time. If such an integer does not exist, then $rank(S, x) = 0$. Otherwise, let $1 \leq j < t$ be the smallest integer such that $q = s_{it+1} + \sum_{h=1}^{j} g_{it+1+h} \geq x$. j can be found by linear search in $\mathcal{O}(t) = \mathcal{O}(\log u)$ time. Then,

$$rank(S, x) = \begin{cases} it + j + 1 \; if \; q = x \\ it + j \qquad if \; q > x \end{cases}$$

The bit-length of C is at most $\mathcal{O}(n \log u)$, so a pointer to C takes $\log n + \log\log u + \mathcal{O}(1) \leq \log u + \log\log u + \mathcal{O}(1)$ bits. It follows that for each block we spend $2\log u + \log\log u + \mathcal{O}(1)$ bits (one element s_{it} and a pointer to C), so the blocks take overall $(2\log u + \log\log u + \mathcal{O}(1)) \cdot n/\log u = 2n + o(n)$ bits. We obtain:

Lemma 3. *Let d be the number of distinct gap lengths between elements in S. The binary-searchable dictionary described in section 3.1 occupies* $(1 + o(1))nH_0(G) + (3 + o(1))n + \mathcal{O}((d + \log\log u)\log u)$ *bits of space and supports rank and select queries in* $\mathcal{O}(\log u)$ *time.*

Note that the size of the proposed BSD can be *exponentially* smaller than u if S is sparse. In the next section we show how to obtain $\mathcal{O}(\log(u/n) + \log\log u)$-time queries without asymptotically increasing space usage.

3.2 A Fully-Indexable Dictionary

Let $v = \lceil u \log^2 u/n \rceil$. The idea is to divide U into blocks of v elements, and store a BSD for each block.

We build a constant-time rank and select succinct bitvector $V[0, ..., \lceil u/v \rceil - 1]$ defined as $V[i] = 1$ if and only if $S \cap \{iv, ..., (i+1)v - 1\} \neq \emptyset$. Additionally, one array $R[0, ..., \lceil u/v \rceil - 1]$ stores sampled *rank* results: $R[0] = 0$ and $R[i] = rank(S, iv - 1)$ for $i > 0$. We build a binary-searchable dictionary $BSD(i)$

[3] We point to $code(g_{it+2})$ instead of $code(g_{it+1})$ because s_{it+1} is explicitly stored. As a matter of fact, we can avoid storing $code(g_{it+1})$ in C.

for each set $S_i = \{x - iv \mid x \in S \cap \{iv, ..., (i+1)v - 1\}\},\ i = 0, ..., \lceil u/v \rceil - 1$, where we use the same codebook D for all the BSD structures (i.e. D is computed according to all gaps $g_1, ..., g_n$). Note that there may exist a set S_i (or more than one) such that its first gap does not belong to $\{g_1, ..., g_n\}$. This happens each time an element s_i is the first of its block $b = \lfloor s_i/v \rfloor > 0$, the gap g_i overlaps blocks b and $b-1$, and $s_i - b \cdot v + 1 \notin \{g_1, ..., g_n\}$. However, by construction of the BSD data structure (see previous section), the first gap in S_i is never used (since we store the smallest element of S_i explicitly), so this event does not affect overall gap frequencies nor space requirements of the array D. Finally, one array $SEL[0, ..., \lceil n/t \rceil - 1]$, where $t = \lceil \log^2 u \rceil$, stores the (number of the) block containing s_{it+1}: $SEL[i] = \lfloor s_{it+1}/v \rfloor$, for $i = 0, ..., \lceil n/t \rceil - 1$.

Using the above described structures, we can now show how to efficiently solve queries. $rank(S, x)$, $x \in U = \{0, ..., u-1\}$, is implemented by accessing the $\lfloor x/v \rfloor$-th block and calling $rank$ on $BSD(\lfloor x/v \rfloor)$. More formally,

$$rank(S, x) = R[\lfloor x/v \rfloor] + rank(S_{\lfloor x/v \rfloor}, x \mod v)$$

where $rank(S_{\lfloor x/v \rfloor}, x \mod v)$ is called on the structure $BSD(\lfloor x/v \rfloor)$. Rank is thus solved in $\mathcal{O}(\log v) = \mathcal{O}(\log(u/n) + \log\log u)$ time. To solve $select(S, i)$, we firstly find by binary search the block containing s_{i+1}, and then call $select$ on the corresponding BSD. More in detail, let $q_l = SEL[\lfloor i/t \rfloor]$ and $q_r = SEL[\lfloor i/t \rfloor + 1]$ if $\lfloor i/t \rfloor + 1 < \lceil n/t \rceil$, $q_r = q_l$ otherwise. By construction of SEL, the block containing element s_{i+1} is one of $q_l, q_l + 1, ..., q_r$. Note that the number $q_r - q_l + 1$ of blocks of interest can be arbitrary large since there may be an arbitrary number of *empty* blocks among them. However, at most t of them will contain *at least* one element (by construction of SEL). Then, we can perform binary search only on the blocks marked with a 1 in the array V: during binary search we access blocks at positions of the form $select(V, j)$ (note: this is a constant-time select performed on the bitvector V), starting with the range $j \in [rank(V, q_l) - 1, rank(V, q_r) - 1]$. Binary search is performed according to partial ranks (array R). Let $q_l \le q_m \le q_r$ be the biggest integer such that $R[q_m] \le i < R[q_m + 1]$ (if $q_m + 1 \ge \lceil u/v \rceil$ then simply ignore the upper bound in the previous inequality). According to the above considerations, q_m can be found in $\mathcal{O}(\log t) = \mathcal{O}(\log\log u)$ time using binary search. We can solve $select(S, i)$ as follows:

$$select(S, i) = q_m \cdot v + select(S_{q_m}, i - R[q_m])$$

where $select(S_{q_m}, i - R[q_m])$ is called on the structure $BSD(q_m)$. $select$ is thus solved on our FID in $\mathcal{O}(\log v) + \mathcal{O}(\log\log u) = \mathcal{O}(\log(u/n) + \log\log u)$ time.

Bitvector V takes $(1 + o(1))u/v = (1 + o(1))n/\log^2 u = o(n)$ bits. Arrays R and SEL take $\log u \cdot u/v = n/\log u = o(n)$ and $\log u \cdot n/t = n/\log u = o(n)$ bits of space, respectively. Finally, all BSD data structures take overall $(1 + o(1))nH_0(G) + (3 + o(1))n$ bits, and the codebook D and the universal tables take $\mathcal{O}((d + \log\log u)\log u)$ bits. We can state our final result:

Theorem 2. *Let d be the number of distinct gap lengths between elements in S. The FID described in section 3.2 takes $(1+o(1))nH_0(G)+(3+o(1))n+\mathcal{O}((d+\log\log u)\log u)$ bits of space and supports rank and select queries in $\mathcal{O}(\log(u/n)+\log\log u)$ time.*

The result stated in Theorem 2 improves the space of [11,17], reducing both leading and $o(u)$ terms from $gap+\mathcal{O}(n\log\log(u/n))$ and $u\log\log u/\log u$ bits to $(1+o(1))nH_0(G)+(3+o(1))n$ and $\mathcal{O}((d+\log\log u)\log u)\subseteq\mathcal{O}(\sqrt{u}\log u)$ bits, respectively. This improvement comes at the price of a $\mathcal{O}(\log(u/n)+\log\log u)$ slowdown in all query times. Notice that we cannot apply the general technique proposed by Mäkinen and Navarro in [11] in order to obtain $\mathcal{O}(1)$ query times since $code()$ does not (always) satisfy $|code(x)|\in\mathcal{O}(\log x)$ (this is one of the properties characterizing *random access self-delimiting codes* [11]). An interesting line of research would be to envision a broader class of codes (including $code()$) for which we can describe a general technique guaranteeing constant-time queries.

4 $H_0(G)$ in practice

In order to assess also in practice the differences between the above discussed measures, we adopted the approach of [8] and simulated several sets, computing for each of them the number of bits per item required by *gap*, $gap+Z_\delta$, $uH_0(S)$, $nH_0(G)$, $nH_0(G)+Z_\delta$, and $nH_0(G)+Z_\delta+CB$, where the last two measures refer to $H_0(G)$ plus the overhead introduced by δ-encoding (i.e. encoding $g_1,...,g_n$ as described in the previous section) and by the codebook size (CB).

Gaps were generated according to uniform (Table 1) and binomial (Table 2) distributions. Table 1 reports the same experiment performed in [8] (except from the facts that we use δ instead of γ and we do not consider RLE), updated with our measure $nH_0(G)$. As expected, in this case $nH_0(G)$ performs slightly worse than *gap* when taking into account all encoding overheads (columns 3 and 7). This can be explained by the fact that gaps are uniform, thus making $gap+Z_\delta$ and $nH_0(G)+Z_\delta$ (without the codebook) almost equivalent. An interesting fact—in accordance with Theorem 1—is that, even this being its worst case, $nH_0(G)$ is always smaller (by about 0.5 bits per item) than $uH_0(S)$.

The advantages of using $nH_0(G)$ become evident when non-uniform distributions are used. Table 2 reports the results on binomially-distributed gaps[4]. As expected, in this case our measure considerably improves on *gap*: if the two techniques are compared while taking into account all encoding overheads (columns 3 and 7), our strategy requires about 58% the space of *gap* encoding.

[4] We chose a binomial distribution in order to model a scenario in which gap lengths are accumulated around a value $\mu\gg 0$ (in this case, μ is the mean). Intuitively, in this case *gap* does not perform well because small numbers are not frequent.

$\log(max\_gap)$	gap	$gap + Z_\delta$	$uH_0(S)$	$nH_0(G)$	$nH_0(G) + Z_\delta$	$nH_0(G) + Z_\delta + CB$
1	1.66717	3.00151	2.00103	1.58496	2.99842	2.99848
2	2.20164	3.80142	2.75854	2.32191	3.79349	3.79364
3	2.77733	5.00151	3.61667	3.16987	4.98418	4.98454
4	3.47452	6.53906	4.5389	4.08735	6.50696	6.50781
5	4.2771	7.79638	5.50097	5.04417	7.75575	7.75773
6	5.15079	8.90439	6.48606	6.02187	8.8685	8.87305
7	6.09095	10.0028	7.4809	7.01044	9.94679	9.95711
8	7.04186	11.9893	8.48908	8.00377	11.889	11.9122
9	8.02066	13.4915	9.50168	8.99923	13.3703	13.4216
10	9.01571	14.7531	10.5266	9.99358	14.5752	14.6879
11	10.0076	15.8755	11.5554	10.9857	15.661	15.9068
12	11.0103	16.9465	12.599	11.9707	16.6565	17.1892
13	12.0031	17.9701	13.6584	12.94	17.5894	18.7364
14	13.0009	18.9844	14.7359	13.8789	18.4625	20.9157
15	13.996	19.9873	15.839	14.7427	19.2538	24.2575

Table 1. Comparison between gap, $gap+Z_\delta$, $uH_0(S)$, $nH_0(G)$, $nH_0(G)+Z_\delta$ (i.e. accounting for the δ overhead per symbol), and $nH_0(G) + Z_\delta + CB$ (i.e. accounting for the δ and codebook CB overhead per symbol) on randomly-generated sets. Gaps between the n items (n affects only the variance of the results; we used $n = 10^5$) are uniformly distributed in the interval $[1, max\_gap]$. All columns except the first report the number of bits per item required by each method.

$\log(max\_gap)$	gap	$gap + Z_\delta$	$uH_0(S)$	$nH_0(G)$	$nH_0(G) + Z_\delta$	$nH_0(G) + Z_\delta + CB$
1	1.74989	3.24967	2.22939	1.50052	2.50156	2.50162
2	2.25085	4.12525	3.16331	2.03377	3.00555	3.0057
3	2.88491	4.94587	4.18044	2.5445	3.49472	3.49508
4	3.77183	7.31887	5.22493	3.04741	4.094	4.09485
5	4.70015	8.69979	6.27376	3.54494	4.82176	4.82326
6	5.64788	9.64788	7.31532	4.04711	5.61441	5.61679
7	6.60309	10.6031	8.3491	4.54742	6.3782	6.3822
8	7.57464	12.7239	9.37466	5.04947	7.08812	7.09424
9	8.55226	14.5523	10.3937	5.54834	7.75208	7.76178
10	9.53716	15.5372	11.4078	6.04518	8.33989	8.35386
11	10.5229	16.5229	12.4178	6.54489	8.93035	8.95219
12	11.516	17.516	13.425	7.04343	9.56187	9.59411
13	12.5135	18.5134	14.4301	7.54485	10.3296	10.3775
14	13.5082	19.5082	15.4338	8.03851	11.1441	11.2149
15	14.5084	20.5084	16.4364	8.53758	11.9996	12.1044

Table 2. Comparison between gap, $gap+Z_\delta$, $uH_0(S)$, $nH_0(G)$, $nH_0(G)+Z_\delta$, and $nH_0(G)+Z_\delta+CB$ on randomly-generated sets. Gaps between the n items ($n = 10^5$) are binomially distributed in the (shifted) interval $[1, max\_gap]$ with success probability $p = 1/2$. All columns except the first report the number of bits per item required by each method.

5 Conclusions

In this paper we introduced $H_0(G)$, a new data-aware measure based on the idea of compressing the gaps between elements of a set $S \subseteq \{0, ..., u-1\}$. We provided new theoretical upper-bounds for this measure, and showed that in practice—if the gap stream is compressible—$H_0(G)$ considerably improves space usage of gap encoding techniques combined with logarithmic codes such as Elias δ-encoding. Finally, we proposed a new zero-order representation of bitvectors based on our new measure and a compressed-gap fully-indexable dictionary supporting fast queries and taking small space in addition to $nH_0(G)$.

As expected, simulations confirmed that the proposed compressed-gap measure is particularly convenient in situations where the gaps follow a non-uniform distribution or they are dominated mainly by large numbers. The main drawback of $nH_0(G)$ seems to be the overhead introduced by the zero-order compressor, which in our solution is of $\Theta(\sqrt{u}\log u)$ bits in the worst case. However, in some practical applications this overhead—being proportional to the number d of *distinct* gap lengths—is expected to be negligible with respect to the overall structure size. One example of such an application is run-length compression of the BWT of highly repetitive text collections (e.g. genome variants), where run lengths are expected to scale linearly with the *number of documents* in the collection [12,18].

We plan to implement our FID and test it against state-of-the-art practical gap-encoded bitvector representations (e.g. `sd_vector` of SDSL[7]). Notice that in practice Huffman-compression of the gaps should be preferred to universal delta-encoding, as the additional overhead is much smaller (i.e. we can remove the $o(nH_0(G))$ term). Our FID could find a first application in repetition-aware self-indexing, e.g. by using it as building block of a more space-efficient run-length compressed suffix array (RLCSA[18]).

References

1. Andersson, A.A., Thorup, M.: Tight (er) worst-case bounds on dynamic searching and priority queues. In: Proceedings of the thirty-second annual ACM symposium on Theory of computing. pp. 335–342. ACM (2000)
2. Beame, P., Fich, F.E.: Optimal bounds for the predecessor problem. In: Proceedings of the thirty-first annual ACM symposium on Theory of computing. pp. 295–304. ACM (1999)
3. Bell, T.C., Moffat, A., Nevill-Manning, C.G., Witten, I.H., Zobel, J.: Data compression in full-text retrieval systems. Journal of the American Society for Information Science 44(9), 508–531 (1993)
4. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Tech. rep., Digital Equipment Corporation (1994)
5. Clark, D.: Compact Pat trees. Ph.D. thesis, University of Waterloo (1996)
6. Elias, P.: Universal codeword sets and representations of the integers. Information Theory, IEEE Transactions on 21(2), 194–203 (1975)
7. Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: Plug and play with succinct data structures. In: 13th International Symposium on Experimental Algorithms, (SEA 2014). pp. 326–337 (2014)

8. Grossi, R., Gupta, A., Vitter, J.S.: When indexing equals compression: Experiments with compressing suffix arrays and applications. In: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms. pp. 636–645. Society for Industrial and Applied Mathematics (2004)
9. Gupta, A., Hon, W.K., Shah, R., Vitter, J.S.: Compressed data structures: Dictionaries and data-aware measures. Theoretical Computer Science 387(3), 313–331 (2007)
10. Jacobson, G.J.: Succinct static data structures. Ph.D. thesis, Dept. of Computer Science, Carnegie Mellon University (1988)
11. Mäkinen, V., Navarro, G.: Rank and select revisited and extended. Theoretical Computer Science 387(3), 332–347 (2007)
12. Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and retrieval of highly repetitive sequence collections. Journal of Computational Biology 17(3), 281–308 (2010)
13. Munro, J.I.: Tables. In: Foundations of Software Technology and Theoretical Computer Science. pp. 37–42. Springer (1996)
14. Pagh, R.: Low redundancy in static dictionaries with constant query time. SIAM Journal on Computing 31(2), 353–363 (2001)
15. Pătraşcu, M.: Succincter. In: Foundations of Computer Science, 2008. FOCS'08. IEEE 49th Annual IEEE Symposium on. pp. 305–313. IEEE (2008)
16. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms. pp. 233–242. Society for Industrial and Applied Mathematics (2002)
17. Sadakane, K., Grossi, R.: Squeezing succinct data structures into entropy bounds. In: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm. pp. 1230–1239. ACM (2006)
18. Sirén, J.: Compressed full-text indexes for highly repetitive collections. Ph.D. thesis, University of Helsinki, Department of Computer Science (2012)